Fundamentals of Artificial Intelligence [H02C1a]

Xinhai Zou (r0727971)

Contents

1 Intro to AI

1.1 Overview

AI starts from **1955** in Bell Telephone Laboratories. What is AI today?

- Planning & Scheduling

- Declarative Programming

- Robotics

- Computer vision

- Machine learning

- language processing

- Human computer interaction

- theorem proving

Sometime, we must make a decision when faced with insufficient information. Being rational means maximizing your expected utility.

1.2 Summary

1. One summer was not enough to shape AI -> but many significant advances have been made since!

2. Movie AI, News AI and Research AI differ

3. Modern AI systems combine 'reasoning' with 'learning' -> this course: 'resoning' focus

4. Much to say about ethics and fairness of AI systems

2 Rational agent

2.1 Overview

- Rational agent

    - Agent: an entity that perceives (from env.) and acts (to env.)
    - Rational: maximizing expected utility
    - Rational agent: selects actions that maximize its (expected) utility,
      i.e., they do the right thing.

- Different environments

    - Accessible vs. inaccessible (fully observable vs. partially observable)
        * environment is fully observed by agent?
    - dterministic vs. stochasitic
        * is the next state of the envioronment completely determined by
          the current state?
    - episodic vs. sequential
        * can the quality of an action be evaluated within **an episode**
          (perception + action)?
    - static vs. dynamic
        * can the environment change while the agent is deliberating?
        * dynamic - not turn-based game, static - mostly turn-based game
    - discrete vs. continuous
        * is the value discrete (nominal) or continuous (numerical)
    - single vs. multi-agent
        * which entities have to regarded as agents? (competitive or coop-
          erative scenarios?)

    Example

- Reflex agents

    - choose (respond) action only based on current state
    - do not consider the future consequences of their actions

- Planning/ Goal based agent

    - choose action based on consequences of actions (future)
    - consider how the world will be

Table 1: Examples using different environments

Env.	Observable	Agents	Deterministic	Episodic	Static	Discrete
Crossword puzzle	Fully	Single	Deterministic	sequential	Static	Discrete
Chess with a clock						
Poker						
Backgammon						
Taxi driving						
Medical diagnosis						
Image analysis						
Part-picking robot						
Refinery controller						
English tutor						
Pac-Mac	Observable	Multi	Stochasitic	Sequential	Dynamic	Discrete

2.2 Summary

1. an agent is something that perceives and acts, being rational is to obtain maximal expected value. An rational agent will take the action that maximizes its expected performance given the percept sequence and its knowledge of the environment.

2. an agent program maps from a percept to an action

   (a) reflex agents respond based on current state

   (b) goal-based agents respond based on current and future state

   (c) learning agents improve their behaviour over time

3. some environments are more demanding than others

3 State space

3.1 Overview

How to represent a problem in machine? For a basic search problem, there are five ingredients.

1. States: represent the states in a clear way

2. Actions: define the actions you can execute and change the world state, normally $\delta : X \rightarrow Y$

3. Initial state: what is the initial state, the **starting** point

4. Goal formulation: define which states you want to end

5. Specification of the search cost: the execution costs

Take Pac-man - path-finding as an example, which is shown below:

- States: (x, y), where x is x-axis and y is y-axis

- Initial state: starting point (x_0, y_0)

- Actions: different actions

  - east: $f_{east} : (x, y) \rightarrow (x + 1, y)$
  - west: $f_{west} : (x, y) \rightarrow (x - 1, y)$
  - north: $f_{north} : (x, y) \rightarrow (x, y + 1)$
  - south: $f_{south} : (x, y) \rightarrow (x, y - 1)$

- Goal test: goal point/state (x^*, y^*)

- Path cost: each step cost 1 step unit, only have <specific number> state unit

Sometimes, we need a check-loop detection. We have two different structure for state-space representation: graph and search tree.

- graph

  - For graph, during searching, we often keep track of the already visited nodes to avoid loops and redundancies.

- search tree

  - nodes show states, each node in the search tree is an entire PATH in the state space graph.

Some key issues:

- definition of state space and transition model

- choose implicit graph or implicit tree

- search forward or backward

- use optimal solution or any solution

- decompose difficult problem to simpler problem (problem reduction/ decomposability)

For forward and backward search, the definition is shown as belwo. The efficiency can be improved based on different branching factor. However, backward is not always possible, e.g., playing chess.

1. Forward: start from initial state and search towards goal

2. Backward: start from the goal state and search toward initial state

3.2 Summary

1. Before an agent can start to perform on searching problem, 5 ingredients are important:

   (a) State
   (b) Initial state
   (c) Goal state
   (d) Action
   (e) Cost

4 Uniformed search

4.1 Overview

Some terminologies for searching problem in graph:

1. node expansion: generating all successor node considering the availble actions (the whole process)

2. explored nodes: nodes that have already been expanded

3. frontier/ fringe: set of all nodes available for expansion (candidate for expansion)

4. search stratege: defines which node is expanded next (the strategies)

4.2 Graph search

Graph search, frontier = <s,a>, <s,b>, its basic algorithm:
Input:
  a graph,
  a set of start nodes,
  Boolean procedure *goal(n)* that tests if n is a goal node.
frontier := $< s >$, s is a start node
while *frontier* is not empty:
  select and **remove** path $< n_0, ..., n_k >$ from *frontier* **for every** neighbor n of n_k
      if *goal(n)*
          return $< n_0, ..., n_k, n >$
      else add $< n_0, ..., n_k, n >$ to *frontier*
end while

For loop detection, the loop-detection algorithm will be:
Input:
  a graph,
  a set of start nodes,
  Boolean procedure *goal(n)* that tests if n is a goal node.
frontier := $< s >$, s is a start node
while *frontier* is not empty:
  select and **remove** path $< n_0, ..., n_k >$ from *frontier* **for every** neighbor n of n_k
      if *goal(n)*
          return $< n_0, ..., n_k, n >$
      else if $n \notin n_0, ..., n_k$
          then add $< n_0, ..., n_k, n >$ to *frontier*
end while

6

Two basic searching strategies for selecting nodes to explore: depth-first search and breadth-first search.

4.3 DFS and BFS

4.3.1 Depth-First Search (DFS)

Its selecting strategy is **LIFO = "Last In First Out"**, we work with **stack**.

- Time complexity: $O(b^m)$

  – with b: branching factor, m: maximum depth

- Space complexity: $O(b \times m)$

  – with b: branching factor, m: maximum depth

The property of DFS:

- Time complexity: $O(b^m)$

- Space complexity: $O(b \times m)$

- Completeness: is complete if finite m and prevent loop

- Optimal: no guarantee, it finds the leftmost solution

4.3.2 Breadth-First Search (BFS)

Its selecting strategy is **FIFO = "First In First Out"**, we work with **queue**.

- Time complexity: $O(b^s)$

  – with b: branching factor, s: shallowest-solution depth

- Space complexity: $O(b^s)$

  – with b: branching factor, s: shallowest-solution depth

- Completeness: no matter loop or not, it is complete

- Optimal: optimal only if **cost between each node = 1**

4.3.3 Comparison of DFS & BFS

DFS uses less memory, but needs loop detection; while BFS use more memory, but does not need loop detection. More details are shown as below:

- Depth-First Search

  – Space is restricted

  – Many solutions exist, solutions are long

- no infinite paths

- Breadth-First Search

  - Space is not a problem
  - Solution containing fewest/shallowest depth

4.4 Iterative Deepening

Iterative Deepening is a combinatino of DFS and BFS, the idea is to get advantages from both DFS (space complexity) and BFS (time complexity, optimal, completeness). Run a DFS with depth limit 0/1/2/3/.... (P.S.:LIFO-stack, FIFO-queue).

Furthermore, the **outter** algorithm:

procedure iterative-deepening (
   Input:
   a graph,
   a set of start nodes,
   Boolean function *goal(n)* that tests if *n* is a goal mode
)
depthlimit = 1
while goal not found **do**
   call depth-limited-search(graph, starting nodes, *goal(n)*, *depthlimit*)
   depthlimit = depthlimit + 1

The **inner** algorithm:

procedure depth-limited-search(
   Input:
   a graph,
   a set of start nodes,
   Boolean procedure *goal(n)* that tests if *n* is a goal node,
   depthlimit: natural number
)
frontier := $< s >$: *s* is a start node
while *frontier* is not empty:
   select and **remove** first path $< n_0, ..., n_k >$ from *frontier*
   if $k < depthlimit$
      for every neighbor *n* for n_k
         if *goal(n)*
            then return $n_0, ..., n_k, n$
         else add $n_0, ..., n_k, n$ to start of *frontier*
end while

The property of iterative deepening (ID) is similar to BFS, the only difference is the memory usage which is better than BFS ($O(b \times s) \geq O(b^s)$). *Time complexity almost is the same as BFS, the only difference is ID needs iterative*

8

reconstruction each limit of 0,1,2,3 needs to start from start node (root) and to rebuild the search tree

- Time complexity: $O(b^s)$

    - with b: branching factor, s: shallowest-solution depth

- Space complexity: $O(b \times s)$

    - with b: branching factor, s: shallowest-solution depth

- Completeness: no matter loop or not, it is complete

- Optimal: optimal only if **cost between each node = 1**

4.5 Bi-directional search

Forward and backward search together. But it is not always possible *because backward is not always known.* The algorithm is shown below:

Input:
   a graph,
   a set of start nodes,
   Boolean procedure *goal(n)* that tests if *n* is a goal node.
$frontier_0 := <s>$: *s* is a start node
$frontier_1 := <g>$: *g* is a goal node
$i := 0;$
while $frontier_1$ and $frontier_0$ not empty:
   $i := i + 1$ mode 2; $j := i + 1$ mod 2; (swap *i* and *j*)
   select and **remove** first path $< n_0, ..., n_k >$ from $frontier_i$
   for every neighbor *n* of n_k in direction *i*
      if *n* occurs in path $< n'_0, ..., n'_l >$ of $frontier_j$ with l the length of n in $frontier_j$
         return solution based on $< n'_0, ..., n'_l >$ and $< n_0, ..., n_k >$
      else add $< n_0, ..., n_k, n >$ to end of $frontier_i$
end while

The property of Bi-directional search:

- Time complexity: $O(2 \times b^{\frac{d}{2}})$

    - with b: branching factor, d:maximum depth

- Space complexity: $O(2 \times b^{\frac{d}{2}})$

    - with b: branching factor, d:maximum depth

- Completeness: yes

- Optimal: yes <- **why?**

4.6 Uniform Cost Search (UCS)

BFS finds the shortest path in terms of number of actions. However, it does not find the least-cost path. We will now consider the cost between 2 nodes.

The formula: $cost(< n_0, n_1, ..., n_k >) = \sum_{i=1}^{k} cost(< n_{i-1}, n_i >)$

SELECT becomes **"priority queue, select element with lowest cost first"**

Two possibilities in graph searching: first version and last version.

1. First version: after selecting first lowest-cost node, done.

2. Last version: after checking every lowest-cost node, then done (more optimal).

Then we can have a further optimization: **Branch-and-bound principle:** when a goal is found along a path with cost C, prune all paths on the frontier with cost > C, where these cost can never be optimal since their cost has already > C.

The properties of UCS are:

The property of Bi-directional search:

- Time complexity: $O(b^{1+\frac{C^*}{\epsilon}})$

    - with b: branching factor, C^*: solution cost, arc cost (cost between 2 neighbor nodes) at least ϵ

- Space complexity: $O(b^{1+\frac{C^*}{\epsilon}})$

    - with b: branching factor, C^*: solution cost, arc cost (cost between 2 neighbor nodes) at least ϵ

- Completeness: yes (minimum arc cost is positive, $\epsilon > 0$)

- Optimal: yes

Remember: UCS explains increasing **cost contours**.

- Pros:

    - Complete and Optimal!

- Cons:

    - Explores option in every "direction", just like BFS

    - No information about goal location, only accumulation of cost (advanced algorithm should have advanced function/info providing)

4.7   Summary

1. The difference between each algorithm: **which node on the frontier will be explored next**.

2. Search algorithm is judged by its properties: **time complexity, space complexity, completeness, optimality**.

5 Informal search

5.1 Overview

Different from uninformed search and informed search.

- Uninformed search

  - use only information on **current path**, that is, from goal to fringe

- Informed search

  - use also **heuristics**, that is, information about how close nodes on the fringe are to a goal

A heuristics is a function that estimates **how close a state is to a goal**, which needs **domain knowledge**

There is different heuristic function: manhattan distance, euclidean distance, ...

5.2 Greedy search

The difference between greedy search and uniformed search is **SELECT = "priority queue, order elements by** $h(n)$, which is called "Best-first search". The difference between greedy search and uniformed cost-sensitive search is:

- Greedy Search uses heuristic to order.

- UCS uses accumulated cost to order.

Is it optimal? No, it just takes **the most obvious result** as next step, but may not be the best one. (Short-term vs. Long-term) Thus, the properties of greedy search is: (Time and space complexity is just badly-guide DFS)

- Time complexity: $O(b^m)$

  - with b: branching factor, m: maximum depth

- Space complexity: $O(b^m)$

  - with b: branching factor, m: maximum depth

- Completeness: no, like badly DFS

- Optimal: no, it just choose the most obvious result instead of the optimal one

12

5.3   Beam search

A variant of ~~greedy search~~ and breadth-first search. It is similar to breadth-first, but only expend *k* times **each tier**. First build the whole tier, then keep *k* elements in frontier. The properties of beam search is:

- Time complexity: $O(k \times b \times s)$

  - with k: hyperparameter, b: branching factor, s: shallowest-solution depth

- Space complexity: $O(k \times b)$

  - with k: hyperparameter, s: shallowest-solution depth

- Completeness: no, may drop the goal

- Optimal: no, not completeness not even optimal

When k = 1, it is called **"Hill-Climbing search"**.

5.4   Combination of UCS and Greedy Search

As we known, UCS uses **path cost = "Backward cost"** = $g(n)$ (Figure 1), and Greedy search uses **heuristic = "Forward cost** = $h(n)$ (Figure 2). And if we combine those two cost to estimate the quality of our current state: $f(n) = g(n) + h(n)$. The **SELECT = "priority queue, order elements by $f(n)$"**
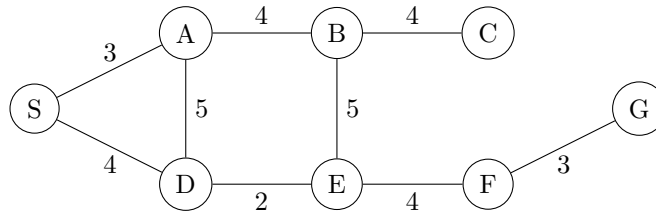

Figure 1: Path cost $g(n)$

However, this is not enough for optimal. We need to do something on the heuristic cost. Otherwises, the actual bad goal cost < estimated good goal cost. **Admissible heuritics** is needed, they **underestimate the actual cost**. Saying that *h* is admissible iff for all nodes $n \leq h(n) \leq h^*(n)$ where $h^*(n)$ is the actual cost to the closest goal.

How to come up with admissible heuristics? The cost of an optimal solution to a **relaxed problem** is admissible heuristics, an understimate, for the original problem.
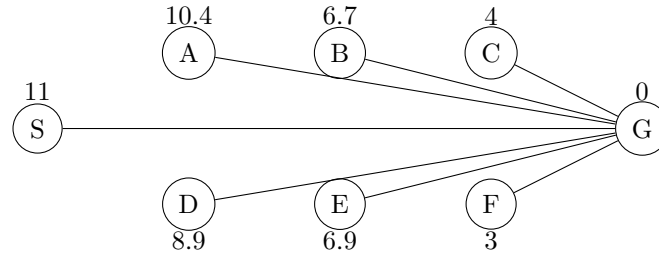
Figure 2: Heuristic $h(n)$

Larger admissible heuristics, better estimation, closer to actual value, but cannot larger than actual value (upper bound). For example, $h2$ dominates $h1$ means for all nodes $n : h2(n) \geq h1(n)$. In this case, $h2$ is always better than $h1$, but for all nodes $h1(n) \leq h2(n) \leq h^*(n)$. For two heuristics $h1$ and $h2$, $h3(n) = max(h1(n), h2(n))$ always dominates both $h1$ and $h2$, because for all nodes n, $h3(n) \geq h1(n)$ and $h3(n) \geq h1(n)$.

5.5 UCS + Greedy Search

Generally, UCS + Greedy = A^* without redundant path elimination. For redundant paths: A^*, for memory usage IDA^*. The properties of combination between UCS and Greedy search is shown:

- Time complexity: $O(b^{1+\frac{C'}{\epsilon}})$ worst case $h = 0$

  - with b: branching factor, C^*: solution cost, arc cost (cost between 2 neighbor nodes) at least ϵ shallowest-solution depth

- Space complexity: $O(b^{1+\frac{C'}{\epsilon}})$ worst case $h = 0$

  - with b: branching factor, C^*: solution cost, arc cost (cost between 2 neighbor nodes) at least ϵ

- Completeness: yes due to optimal

- Optimal: yes, if admissible heuristics

Proof of optimality of UCS + G is shown:

1. Imagine B (suboptimal) is on the fringe

2. Imagine some ancestor n of A (optimal) is on the fringe too

3. n will expanded before B

   (a) $f(n)$ is less or equal to $f(A)$ due to admissible heuristics
   (b) $f(A)$ is less or equal to $f(B)$ due to admissble heuristics

14

(c) n expands before B

4. All ancestors of A expand before B

5. A (optimal) expands before B (suboptimal)

6. UCS + G search is optimal

5.6 Redundant Paths

Key: failure to detect **repeated states** can cause exponentially more work.
Idea: never expand a state twice. One way to implement is shown:

1. **Keep track** of set of expanded states "closed set"

2. Expand the search tree node-by-node

3. Before expanding a node, **check** to make sure its state has never been expanded before

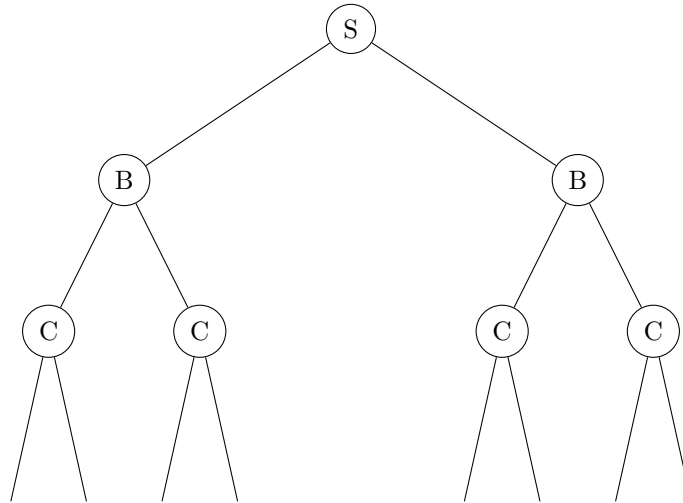4. **If not new, skip it, if new add to closed set.**

Figure 3: Repeated state without redundant path elimination

Generally,

- UCS + Greedy search = A^* without redundant path elimination

- A^* = UCS + Greedy search + redundant path elimination

15

The algorithm of redundant path elimination *(First version: the closed set, not the path deletion version)* is shown as below:

Input:
    a graph,
    a set of start nodes,
    Boolean procedure $goal(n)$ that tests if n is a goal node.
frontier $:= < s >$: s is a start node
closed $:= \{\}$
while *frontier* is not empty:
    select and **remove** path $< n_0, ..., n_k >$ from *frontier*
    if $n_k \notin closed$ **then**
        add n_k to *closed*
        if $goal(n_k)$
            return $n_0, ..., n_k$
        for every neighbor n of n_k
            add $n_0, ..., n_k$ to *frontier*
end while

5.7 A^* went wrong

Problem: When A^* gone wrong?. If the heuristics is **not consistent**. There are two options: consistent heuristics or repair the frontier

1. Consistent heuristics: ensure that the first "selected" path to a node always has lowest cost, thus, we can directly prune node by closed set without considering. *(This can only work for consistent admissble heuristics)*

2. Repair the frontier: if there is a path $p = < s, ..., n, ..., m >$ on the frontier and a path p' to n such that $g(p' \geq g(p))$ then remove p' from the frontier. *(This can work for any admissible heuristics)*

5.7.1 Consistent heuristics

Difference between admissibility and consistency. But the main idea for both is "estimated heuristic cost $\leq$ actual consts". Generally, **consistency $\rightarrow$ admissibility**. Consistency makes A^* having a **f-contour**, and increase this **"f-contour"** one by one.

- Admissibility: from A to G, heuristic cost $\leq$ actual cost to goal (for whole)

    - $h(A) \leq$ actual cost from A to G

- Consistency: for each path from power set of from A to G, heuristic "arc" cost $\leq$ actual cost for each arc

    - $h(A) - h(C) \leq$ cost (A to C)

16

5.7.2 repair frontier

For repairing the frontier, we need to compare new repeated path, and the state in the current path, the algorithm is shown as below: **(no closed set exist)**

Input:
    a graph,
    a set of start nodes,
    Boolean procedure $goal(n)$ that tests if n is a goal node.
frontier $:= <s>$: s is a start node
~~*closed* $:= \{\}$~~
while *frontier* is not empty:
    select and **remove** path $<n_0, ..., n_k>$ from *frontier*
    ~~**if** $n_k \notin closed$ **then**~~
        ~~add n_k to *closed*~~
    if $goal(n_k)$
        return $n_0, ..., n_k$
    for every neighbor n of n_k
        add $n_0, ..., n_k$ to *frontier*
    if there are $p = <s, t_1, t_2, ..., t_k, i, ..., m>$ and $p' = <s, s_1, s_2, ..., s_l, i>$ on the
frontier and $g(p') \geq g(p)$
        then remove all such p'
end while

5.8 Iterative Deepening A^*

Idea from A^* with consistent heuristics, with increasing "f-contour". Can we do this **iteratively**? Get DFSś space advantages with A^* time. The difference between ID and IDA^* is **how to choose "contour limit"**. The IDA^* usually combined with loop breaking instead of redundant path elimination (this is not A^*), as this requires to store all generated nodes (closed set).

- For ID, just do a DFS, if no solution, just limit = limit + 1

- For IDA^*, if no solution, **use the smallest f value as the next contour.**

The algorithm of IDA^* is shown as below:

procedure f-limited-search (
    input: a graph,
    a set of start nodes,
    Boolean procedure $goal(n)$ that tests if n is a goal node,
    f_{bound}: positive number
)
frontier $:= <s>$: s is a start node
$f_{next} := \infty$

while *frontier* is not empty:

    select and **remove** first $p = < n_0, ..., n_k >$ from *frontier*

        if $goal(n_k)$ **then return** p

        for every neighbor n of n_k

            compute $f_n = f(< n_0, ..., n_k, n >)$

            if $f_n \leq f_{bound}$

                then add $< n_0, ..., n_k, n >$ to front of *frontier*

            if $f_n > f_{found}$ and $f < f_{next}$

                then $f_{next} := f_n$

end while

return f_{next}

    The properties of IDA^* is shown as below:

- Time complexity: worst case like $b^0 + (b^0 + b^1) + (b^0 + b^1 + b^2) + ... + (b^0 + b^1 + ... + b^N) = ...$, which is $O(b^{\frac{C^*}{\epsilon}})$, as ID is $O(b^d)$

  - with b: branching factor, C^*: solution cost, arc cost (cost between 2 neighbor nodes) at least ϵ shallowest-solution depth, s: shallowest-solution depth

- Space complexity: $O(b \times \frac{C^*}{\epsilon})$ worst case $h = 0$, as ID is $O(b \times s)$

  - with b: branching factor, C^*: solution cost, arc cost (cost between 2 neighbor nodes) at least ϵ, s: shallowest-solution depth

- Completeness: yes due to optimal

- Optimal: yes, under same situation as UCS + Greedy Search *(Consistent admissible heuristics or redundant path elimination)*

5.9 Summary

1. The best heuristics $f(n) = g(n) + h(n)$ is an **admissble heuristics** with the combination of Uniformed Cost Search and Greedy Search.

2. Coming up with a good admissible heuristics is important (larger -> better -> closer to the $h^*(n)$ (actual value), but also should be (for all nodes n) smaller than $h^*(n)$)

3. **Heuristics design** is key for searching problem!

4. Uniformed search vs. Informed Search = with or without **"heuristics"**

5. $f(n) = g(n) + h(n)$, **difference choices $\rightarrow$ difference algorithms**, where without $g(n)$ is Greedy search, and without $h(n)$ is UCS, with both $g(n)$ and $h(n)$ is A^* without redundant path elimination.

6. Important **FOUR** properties for an algorithm: Time complexity, Space complexity, Completeness, Optimality.

6 CSP

6.1 Overview

Two different tasking: planning and identification

1. Planning: sequences of actions: path is important

2. Identification: assignments to variables: goal is important (CSP)

What is CSP?

- A special subset of search problems

- State is defined by varaible X_i with values from a domain D

- Goal is a set of **constraints** specifying allowable combination of values for subsets of variables

For example, the N-Queens problem:

- Variables: $X_{i,j}$

- Domains: $D = 0, 1$

- Constraints: **(not allow touch together (1,1))**

    - $\forall i, j, k (X_{i,j}, X_{i,k} \in (0,0), (0,1), (1,0))$
    - $\forall i, j, k (X_{i,j}, X_{k,j} \in (0,0), (0,1), (1,0))$
    - $\forall i, j, k (X_{i,j}, X_{i+k,j+k} \in (0,0), (0,1), (1,0))$
    - $\forall i, j, k (X_{i,j}, X_{i+k,j-k} \in (0,0), (0,1), (1,0))$
    - $\sum_{i,j} X_{i,j} = N$

6.2 Variaties of CSP

Variables can be classified as: **discrete variables** and **continuous variables**. And the discrete variable can be further classified as **finite domain** and **infinite domains**.
Constrains can be classified as **hard constraints** and **soft constraints**. Hard constraint includes **unary constraint** involing a single variable: $SA \neq green$, or **binary constraint** involing pairs of variable: $SA \neq WA$, or even **higher-order constraints** involing 3 or more variables: $O+O = R+10\times X_1$. **Preferences** (soft constraint) is also important, which will be used in Constrained Optimization Problem.

6.3 Solution to CSP

The standard search formulation is defined as below (initial → assigning values to unassigned varaibles → check goal test):

1. Initial state: the empty assignment, {}

2. Successor funtion: assign a value to an unassigned variables

3. Goal test: the current assignment is complete and satisfies all constraints

6.4 Backtracking

Backtracking search is the basic uniformed algorithm for solving CSPs. DSP with two improvements below is called "backtracking search". In CSP, DFS is better than BFS since BFS will check layer by layer which is unnecessary.

1. One variable at time - only consider assignments to a single variable at each step

2. Check constraints as you go - consider only values which do not conflict previous assignments

The algorithm of backtracking search is shown as below:
function BACKTRACKING-SEARCH(csp) **returns** solution/ failure
    return RECURSIVE-BACKTRACKING({},csp)
function RECURSIVE-BACKTRACKING(assignment, csp) **return** solution/ failure
    if assignment is complete **then return** assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLE[csp], assignment, csp)
    for each value **in** ORDER-DOMAIN-VALUES(var, assignment, csp) **do**
        if value is consistent with assignment given CONSTRAINTS[csp] **then**
            add {var = value} to assignment
            var ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure
                then return result *% Find answer, so return result*
                else remove {var = value} from assignment
    return failure *% No completed solution found, so return failure*

As we can see above, Backtracking = DFS + variable-ordering *(select one variable)* + fail-on-violation *(check constraint)*

6.5 Improvement on Backtracking

The improvement can be applied on three aspects: Ordering, Filtering and Structure:

1. Ordering - which should be next, similar to heuristics

2. Filtering - can we detect inevitable failure in advance?

3. Structure - can we exploit the problem structure?

6.5.1 Filtering

Filtering: keep track of domains for unassigned variables and cross off bad options

Forward checking (FC): Corss off values that violate a constraint when added to the existing assignment; but not provide early detection for all failures. **Assign values to unassigned variables to check, and backtrack. Constraint propagation**: reason from constraint to constraint.

Consistency of a single arc (AC-3): An arc X → Y is consistent iff for *every* x in the tail there is *some* y in the head which could be assigned without violating the constraint. Important: if X loses a value, neighbors of X need to be rechecked! Arc consistency detects failure earlier than forward checking. It checks **all** arcs are consistency once time.

Both constraint propagation and consistency of a single arc can be run as a **preprocessor** or **after each assignment**.

The algorithm is shown as below:

function AC-3(csp) **returns** false if an inconsistency is found and true otherwises

    queue ← a queue of arcs, initially all the arcs in csp *% arcs: double direction (X_i, X_j) and (X_j, X_i) are different*

    while queue is not empty **do**

        (X_i, X_j) ← POP(queue)

        if REVISE(csp, X_i, X_j) **then**

            if size of $D_i = 0$ **then return** false

            for each X_k **in** X_i.**NEIGHBORS - {X_j} do**

                add (X_k, X_i) to queue

    return true

function REVISE(csp, X_i, X_j) **returns** true iff we revise the domain of X_i

    revised ← false

    for each x **in** D_i **do**

        if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j **then**

            delete x from D_i

            revised ← true

    return revised

Difference between forward checking and arc consistency: forward check is global and arc consistency is local (may not detect failure as more as forward check (the global)). More further comparison is shown as below:

- Filtering with Forward Checking (FC)

– after assinging one variable, check the constraints that involve this variables

- Filtering with Arc Consistency (AC3)

    – like FC, but if a variable changed, also check all constraints involing this variable (except itself)

6.5.2 Ordering

There are two types of ordering, as shown below. But there is a common principle: **fail-first**.

1. Variable ordering - which one to assign first?

    (a) Minimum Remaining Values

    (b) Most Constratining Variables

2. Value ordering - which value to try first?

    (a) Least Constraining Value

Minimum Remaining Values: choose the variable with the fewest legal left values in its domain → less values means less steps to iterate its remaining values → easy to check whether fail or success, fail-first. It is also called "Most Constrained Variable"

Most Constraining Variables: choose the variable with the most constraints (only counting constraints involving other unassigned variables) → most constraints → easier to fail → fail-first. It is also called "max degree heuristics"

Least Constraining Value: given a choice of variable, choose the least constraining value.Least value → closer to threshold → easier to delete. ← My guessing. Why least rather than most? You can do this with most, actually.

6.5.3 Structure

Independent substructure: Some structure can be broken into subproblems, which can significantly improve the efficiency. Similar to Bi-direction search, from $O(b^m)$ to $O(2 \times b^{\frac{m}{2}})$.
Value symmetry: in many CSP problems, there are many symmetries, we can reduce the search space by using **symmetry breaking constraints.**

6.6 Constraint Optimization

COP = Constraint Optimization Problem. It uses "branch-and-bound", similar to uniformed search.
Key idea:

1. maintain the 'best' objective value so far

2. constrain solutions to be better than the 'best'

6.7   Summary

1. CSPs are a special kind of search problem

   (a) States are partial assignments

   (b) Goal test defined by constraints

2. Basic solution: backtracking search

3. Speed-up

   (a) Filtering - FC, AC3

   (b) Ordering - Minimum Remaining Values, Most Constraining Variables, Least Constraining Value

   (c) Structure - substructure, symmetry breaking constraints

   (d) Optimization - branch-and-bound

7 Pattern mining

7.1 Overview

Is to find interesting and actionable patterns and models in the data.

Example: Recommender System
Terminology of datasets:

- Itemsets

    - the special case where all attributes are boolean
    - a set of possible items $\mathcal{I}$
    - each example $e \subseteq \mathcal{I}$
    - each hypothesis $h \subseteq \mathcal{I}$
    - hypothesis h covers example e iff $h \subseteq e$
    - h_1 is more general than h_2 iff $h_1 \subseteq h_2$, h_1 has less specific items, which means it is less specific
    - **hypothesis = pattern**

7.2 Frequent itemset mining

Given:
    a set of possible item $\mathcal{I} = i_1, ..., i_n$,
    a dataset D
    a threshold c
Find all itemsets $I \subseteq \mathcal{I}$ such that I is frequent in D, i.e., $freq(I, D) \geq c$.

"covers":

- h_1 covers h_2 means h_1 is more general than h_2, i.e., {s,b,c} covers {s,b,c}, {s,b,c,m}

- more examples, {s,b} covers {s,b}, {s,b,c}, {s,b,c,m}

7.3 Search space

Adding more definition on *Itemsets*

- most general element: {}

- most specific element: $\mathcal{I}$

As we see from Figure 4 and Table 2, if the minimal freq threshold $c = 1$, $D = \{< s, b >\}$

- $freq(\{\}, D) = 1$

- $freq(\{s\}, D) = 1$

- $freq(\{b\}, D) = 0$

- $freq(\{s, b\}, D) = 0$

For minimal freq threshold $c = 1$, the answer would be **{}** and **{s}**.

Table 2: Table for generality ordering lattice for {s,b}
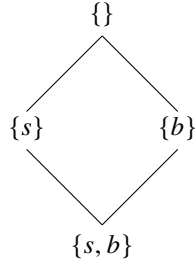
s	b
True	False

Figure 4: Figure for generality ordering lattice for {s,b}

"Anti-monotonicity" principle: if $I_1 \subseteq I_2$ (that is, I_1 is more general than I_2 then $freq(I_1, D) \geq freq(I_2, D)$, since it has more supersets)

There are two borders: specific borders and general borders.

1. S-set = $\{I \subseteq \mathscr{I} | I$ satisfies min and max frequency thresholds, and there is no $J \subseteq \mathscr{I}$ that is strictly more specific than I and also satisfies the thresholds$\}$. $\rightarrow$ meaning that **S is the most specific solution**, it contains all maximall solutions, solution that cannot be extended with extra items.

2. G-set = $\{I \subseteq \mathscr{I} | I$ satisfies min and max frequency thresholds, and there is no $J \subseteq \mathscr{I}$ that is strictly more geeneral than $I\}$. $\rightarrow$ meaning that **G is the most general solution**, it contains all minimal solutions, solution that cannot be deleted.

Concept-learning = 100% frequency on the positives and 0% frequency on the negatives.

7.4 Enumeration Algorithms

The naive algorithm for frequent itemset mining is shown as below:

Input:
    a set of transactions D,
    a set of items $\mathscr{I}$,
    a frequency threshold c.
Queue = {{}} *% Empty set is also a itemset*
for all $I \in Queue$ **do**
    if $freq(I, D) \geq c$
        then output I
        add all $I \cup \{i\}$ to Queue (with $i \in \mathscr{I} - I$) *% Add, s,c,b,m to {} to make {s}, {c}, {b}, {m}*
end for

Improvement: the naive algorithm has too many redundant examples, "lexicographic order" can be used to improve the efficiency. Symmetry breaking: impose the order $s > m > c > b$, and never add an element to a set I if it smaller than the smallest element in the set. *Every set is only generated once and we get a tree instead of a graph.* The algorithm is shown as below (**min + max + lexicographic**):

Input:
    a set of pos transactions *Pos*,
    a set of neg transactions *Neg*,
    a set of items $\mathscr{I}$,
    frequency thresholds t_1, t_2
Queue = {{}}
for all $I \in Queue$ **do**
    if $freq(I, Pos) > t_1$
        then if $freq(I, Neg) \leq t_2$ *% This Neg can be Pos, too, but the truth meaning is $t' \geq freq(I, Pos) \leq t''$*
            then output I
            add all $I \cup \{i\}$ to Queue (with $i \in \mathscr{I} - I$ and i lexicographically smaller than elements of I)
end for

7.5 Summary

1. The basic concept-learning - frequent itemsets mining

2. min + max is just ont type of constraints

3. 100% frequency on positives and 0% on negatives = concept-learning with version spaces

4. Frequent patterns can be turned in association rules to make predictions - similar to Naive Bayes (Probabilistic Machine Learning)

8 Game tree

8.1 Games

There are different kind of games:

- Deterministic or stochastic game

- Multi-player or single player game

- Zero-sum or general game

- Board game ← we will focus on this!

A good game program/ strategy should have the properties that

1. delete **irrelevant branches** of the search tree

2. use **goof evaluation functions** for in-between states

3. **look ahead** as many moves as possible

8.2 Minimax

8.2.1 Terminology

Soem terminology of two-person board games:

- Players are **min** and **max**, where max begins

- **Initial** position

- **Operators**, actions

- **Game tree** is the search tree generated from the possible moves

- **Termination** test, determines when the game is over and what the **value of final state** is

- **Strategy**

8.2.2 Algorithm

For minimax algorithm, we use DFS to generate the game tree. We will apply **utility function**/ heuristics function to each terminal state. During the minimax, max should have the highest value, and min will always choose the worst-value step for max (they are opponents!). The algorithm is shown as below:

function MINIMAX-SEARCH(game, state) **returns** an action
    player ← game.TO-MOVE(state)
    value,move ← MAX-VALUE(game,state)
    return move

27

function MAX-VALUE(game,state) **returns** a (utility,move) pair
　　if game.IS-TERMINAL(state) **then return** game.UTILITY(state,player),null
　　v ← −∞
　　for each *a* **in** game.ACTIONS(state) **do**
　　　　v2,a2 ← MIN-VALUE(game,game.RESULT(state,a))
　　　　if v2 > v **then**
　　　　　　v,move ← v2,a
　　return v,move

function MIN-VALUE(game,state) **returns** a (utility,move) pair
　　if game.IS-TERMINAL(state) **then return** game.UTILITY(state,player),null
　　v ← +∞
　　for each *a* **in** game.ACTION(state) **do**
　　　　v2,a2 ← MAX-VALUE(game,game.RESULT(state,a))
　　　　if v2 < v **then**
　　　　　　v,move ← v2,a
　　return v,move

8.2.3　Evaluation function

The evaluation function is important! Some examples are shown below:

- Material value: pawn/1, knight/3, queen/9

- Other: king safety, good pawn structure

- Rule of thumb: three-point advantage

The preffered evaluation functions are **weighted, linear functions**: $w_1 f_1 + w_2 f_2 + ... + w_n f n$, where w_i are the weights and the f_i are the features of the board. However, it has a strong assumption that all of features are **independent**, which is unrealistic.

8.2.4　Stopping criteria

Stopping critiera can be fixed-depth or iterative deepening (when time is over.) However, only stop and evaluate in **quiescent positions** that will not cause large fluctuations. Because we do not know what happen if so large fluctuations happen may cause → **horizon effect:** try to delay lose but fail and with more cost.

8.3　Alpha-Beta Search

It is an optimization of minimax algorithm, which helps improve its efficiency. Instead of generating tree → propogating the values upwards in the tree. We try to **interleave these two steps**, meaning that exploit → propogate → prune → exploit → propogate → prune. Some terminology is shown as below:

- The temporary values at MAX-nodes are alpha(α) values

- The temporary values at MIN-nodes are beta(β) values

Minimax algorithm with DFS, it provides the **same result** as the complex minimax search to the same depth because only irrelevant nodes are eliminated.

- ALPHA = the value of the best choice we have found so far at any choice point along the path for MAX

    - ALPHA value can **never devrease**

- BETA = the value of the best choice we have found so far at any choice point along the par for MIN

    - BETA value can **never increase**

The Alpha-Beta (pruning) principle is shown as below:

- If an **ALPHA-value is larger or equal than the BETA-value** of a descendant node: stop generation of the children of the descentdant

- If a **BETA-value is smaller or equal than the ALPHA-value** of a descendant node: stop generation of the children of the descendant

The algorithm for ALPHA-BETA is shown as below:

function MINIMAX-SEARCH(game,state) **returns** an action
    player ← game.TO-MOVE(state)
    value,move ← MAX-VALUE(game,state,$-\infty$,$+\infty$)
    return move

function MAX-VALUE(game,state,α,β) **returns** a (utility,move) pair
    if game.IS-TERMINAL(state) **then return** game.UTILITY(state,player),null
    v ← $-\infty$
    for each a **in** game.ACTIONS(state) **do**
        v2,a2 ← MIN-VALUE(game,game.RESULT(state,a),α,β)
        if v2 > v **then**
            v,move ← v2,a
            α ← MAX(α,v) *% prepare α for MIN*
        if v > β **then reutrn** v,move **then return** result *% ALPHA-BETA pruning, just return*
    return v,move

function MIN-VALUE(game,state,α,β) **returns** a (utility,move) pair
    if game.IS-TERMINAL(state) **then return** game.UTILITY(state,player),null

v ← +∞
for each a **in** game.ACTION(state) **do**
    v2,a2 ← MAX-VALUE(game,game.RESULT(state,a),α,β)
    if v2 < v **then**
        v,move ← v2,a
        β ← MIN(β,v) *% prepare β for MAX*
    if v ≤ α **then return** v,move *% ALPHA-BETA pruning, just return*
return v,move

The Gain: best case. If at every layer: the best node is the left-most one **(perfectly ordered tree)**. In the best case, the search expenditure is reduced to $O(b^{\frac{d}{2}})$

8.4 Expectiminimax

If there is posibility, stochastic instead of deterministic.
d_i: possible dice roll
$P(d_i)$: probability of obtaining that roll
$EXPECTIMINIMAX(s) =$
   $UTILITY(s)$ if $IsTerminal(s)$
   $max_a EXPECTIMINIMAX(Result(s,a))$ if $ToMove(s) = MAX$
   $min_a EXPECTIMINIMAX(Result(s,a))$ if $ToMove(s) = MIN$
   $\sum_d P(d) \times EXPECTIMINIMAX(Result(s,d))$ if $ToMove(s) = CHANCE$
$Result(s,d)$: attainable positions from C with d

8.5 Summary

1. A **game** can be defined by the initial state, the operator/actions, a terminal test, a utility function/ outcome of the game

2. In two-player board game, **minimax algorithm** can be used.

3. The **alpha-beta algorithm** produces the **same result** but it is more efficient because it just prunes away irrelevant branches.

4. **Utility/heuristics** of some states will be determined by an **evaluation function**.

5. Games of chance can be handled by some extension, EXPECTIMINIMAX.

6. The success for different games is based on quite different methodologies.

9 Planing

10 MDP